CAPÍTULO 7

SINCRONIZAÇÃO E COMUNICAÇÃO ENTRE PROCESSOS

Prezado(a) aluno(a),

Este capítulo apresenta como a concorrência de processos pode ser implementada, os problemas do compartilhamento de recursos e a solução por meio de semáforo - um dos mecanismos mais usados do sistema operacional para sincronizar processos. Ao final do capítulo, também são brevemente apresentados o problema do deadlock, suas consequências e análise de soluções.

Procure entender bem o problema do compartilhamento de recursos, sabendo que ele pode ser evitado por meio de exclusão mútua. Importante também é compreender como o mecanismo de semáforos pode ser utilizado para implementar a exclusão mútua. A exclusão mútua de dois ou mais recursos compartilhados pode gerar um outro problema: o deadlock, que pode ser evitado por construção, de forma algorítmica.

Bom estudo!

Fala Professor

7.1. Introdução

Na década de 1960, com o surgimento dos sistemas multiprogramáveis, passou a ser possível estruturar aplicações, de modo que partes diferentes do código do programa pudessem executar concorrentemente. Este tipo de aplicação, denominado aplicação concorrente, tem como base a execução cooperativa de múltiplos processos ou *threads*, que trabalham em uma mesma tarefa na busca de um resultado comum.

Em um sistema multiprogramável com um único processador, os processos alternam sua execução, segundo critérios de escalonamento estabelecidos pelo sistema operacional. Mesmo não havendo, nesse tipo de sistema um paralelismo na execução de instruções, uma aplicação concorrente pode obter melhorias no seu desempenho. Em sistemas com múltiplos processadores, a possibilidade do paralelismo na execução de instruções somente estende as vantagens que a programação concorrente proporciona.

É natural que processos de uma aplicação concorrente compartilhem recursos do sistema, como arquivos, dispositivos de E/S e áreas de memória. O compartilhamento de recursos entre processos pode ocasionar situações indesejáveis, capazes até de comprometer a execução das aplicações. Para evitar esse tipo de problema, os processos concorrentes devem ter suas execuções sincronizadas, a partir de mecanismos oferecidos pelo sistema operacional, com o objetivo de garantir o processamento correto dos programas.

7.2. Aplicações concorrentes

Aplicações concorrentes muitas vezes necessitam que os processos comuniquem-se entre si. Esta comunicação pode ser implementada por meio de diversos mecanismos, como variáveis compartilhadas na memória principal ou trocas de mensagens. Nessa situação, é necessário que os processos concorrentes tenham sua execução sincronizada, através de mecanismos do sistema operacional.

Por exemplo, imagine que dois processos concorrentes compartilhem um buffer para trocar informações, por meio de operações de gravação e leitura. Nesse exemplo, um processo só poderá gravar dados no buffer, caso este não esteja cheio. Da mesma forma, um processo só poderá ler dados armazenados do buffer caso exista algum dado para ser lido. Em ambas as situações, os processos deverão aguardar até que o buffer esteja pronto para as operações, seja de gravação, seja de leitura.

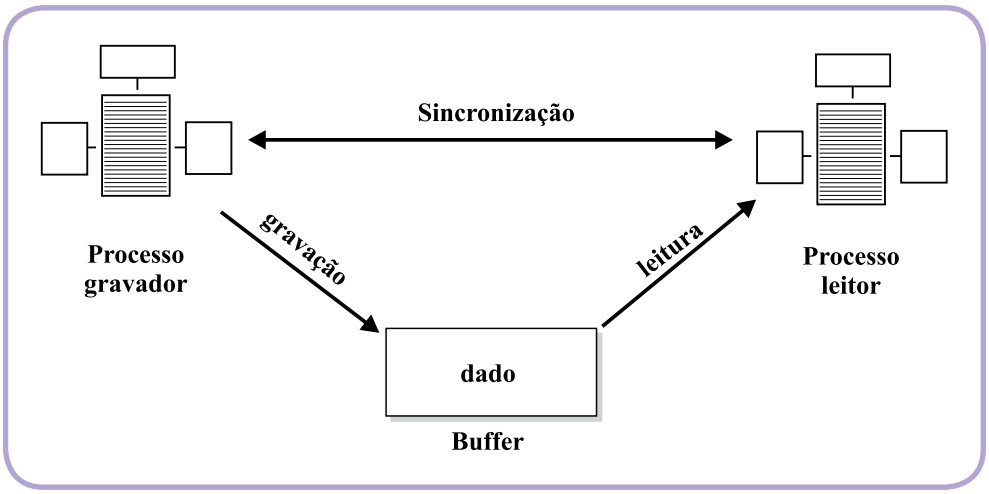

Figura 7-52: Sincronização e comunicação entre processos
Fonte: [1] – Machado e Maia, 2004. Adaptação.

Os mecanismos que garantem a comunicação entre os processos concorrentes e o acesso a recursos compartilhados são chamados **mecanismos de sincronização**. No projeto de sistemas operacionais multipro-

gramáveis, é fundamental a implementação desses mecanismos, para se garantir a integridade e a confiabilidade na execução de aplicações concorrentes [1].

Apesar de o termo processo ser utilizado na exemplificação de aplicações concorrentes, os termos subprocesso e *thread* têm o mesmo significado nessa abordagem.

7.3 Especificação da concorrência em programas

Existem várias notações utilizadas para especificar a concorrência em programas, ou seja, as partes de um programa que devem ser executadas concorrentemente. As técnicas mais recentes tentam expressar a concorrência no código dos programas de uma forma clara e estruturada [1].

A primeira notação para a especificação da concorrência em um programa foram os comandos FORK e JOIN, criados por Conway (1963) e Dennis e Van Horn (1966). Nessa notação, um processo, ao executar o comando FORK, cria um novo processo que executa concorrentemente. O comando JOIN, ao ser executado, faz com que o processo criador sincronize com o processo criado. Em outras palavras, o processo ficará aguardando, até que o que foi criado termine [1].

Uma das implementações mais claras e simples de expressar a concorrência em um programa é a utilização dos comandos PARBEGIN e PAREND, propostas por Dijkstra, em 1965. O comando PARBEGIN especifica que a sequência de comandos seja executada concorrentemente em uma ordem imprevisível, através da criação de um processo para cada comando. O comando PAREND define um ponto de sincronização, a partir de onde o processamento continuará somente quando todos os processos ou *threads* criados tiverem terminado suas execuções.

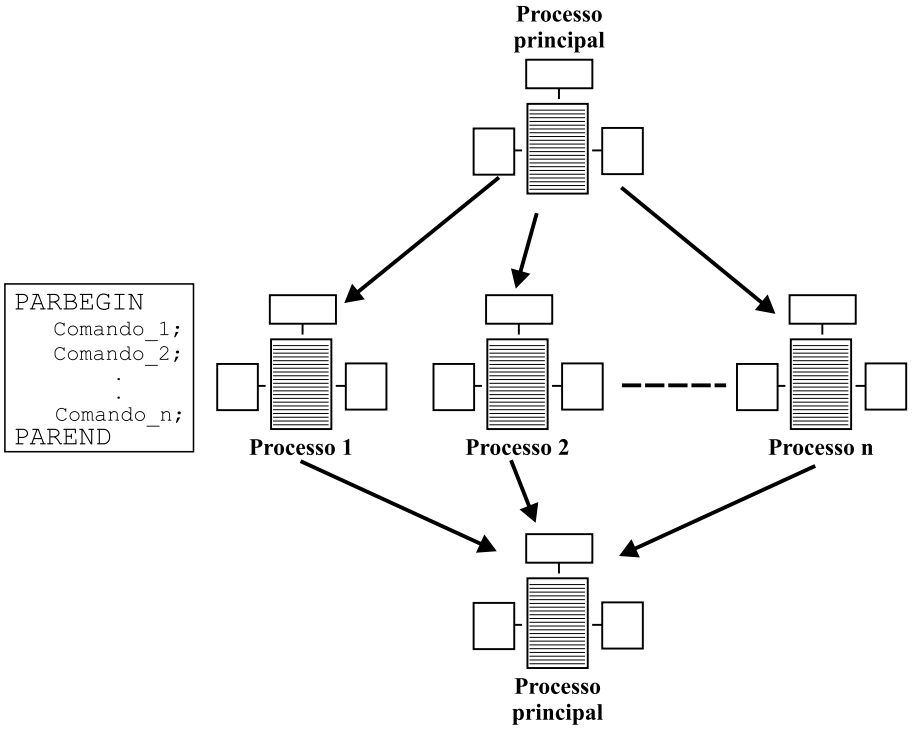

Figura 7-53: Especificação da concorrência em programas
Fonte: [1] – Machado e Maia, 2004. Adaptação.

7.4. Problemas de compartilhamento de recursos

Para a compreensão de como a sincronização entre processos concorrentes é fundamental para a confiabilidade dos sistemas multiprogramáveis, são apresentados alguns problemas de compartilhamento de recursos. A primeira situação envolve o compartilhamento de um arquivo em disco; a segunda apresenta uma variável na memória principal sendo compartilhada por dois processos.

O primeiro problema [1] é analisado a partir do programa "Conta_Corrente", que atualiza o saldo bancário de um cliente após um lançamento de débito ou crédito no arquivo de contas-correntes Arq_Contas. Nesse arquivo são armazenados os saldos de todos os correntistas do banco. O programa lê o registro do cliente no arquivo (Reg_Cliente), lê o valor a ser depositado ou retirado (Valor_Dep_Ret) e, em seguida, atualiza o saldo no arquivo de contas.

```
PROGRAMA Conta_Corrente;
      .
      .
      LEIA (Arq_Contas, Reg_Cliente);
      LEIALN (Valor_Dep_Ret);
      Reg_Cliente.Saldo   :=   Reg_Cliente.Saldo   +
Valor_Dep_Ret;
      ESCREVA (Arq_Contas, Reg_Cliente);
      .
      .
FIM.
```

Para analisar a situação de compartilhamento de recursos, considere o exemplo em que dois processos concorrentes – pertencentes a dois funcionários do banco – atualizam o saldo de um mesmo cliente simultaneamente. Suponha que o processo do primeiro funcionário (Caixa 1) leia o registro do cliente e some ao campo Saldo o valor do lançamento de crédito. Porém, antes de gravar o novo saldo no arquivo, o processo do segundo funcionário (Caixa 2) leia o registro do mesmo cliente, para realizar outro lançamento, desta vez de débito. Independentemente de qual dos processos atualize primeiro o saldo no arquivo, o dado gravado estará inconsistente.

Esse problema ocorre porque o segundo processo realiza acesso à variável compartilhada (saldo do cliente - Reg_Cliente.Saldo) simultaneamente ao primeiro processo. Devido a isso, o segundo processo leu um valor nessa variável que ainda não foi atualizado pelo primeiro processo. Para evitar esse problema, o segundo processo deveria aguardar até que o primeiro terminasse seu acesso à variável compartilhada, ao invés de acessá-la concorrente.

Já imaginou se as agências bancárias não tivessem mecanismos para impedir os problemas relacionados ao acesso simultâneo aos recursos compartilhados?

Um outro exemplo [1], ainda mais simples, em que o problema da concorrência pode levar a resultados inesperados, é a situação em que dois processos (A e B) executam um comando de atribuição. Suponha que a variável X seja inicializada com o valor 2, que o Processo A some 1 à variável X e o Processo B subtraia 1 da mesma variável que está sendo compartilhada.

Processo A	Processo B
x := x + 1;	x := x - 1;

Seria razoável pensar que o resultado final da variável X, após a execução dos Processos A e B, continuasse inalterado, porém, isso nem sempre será verdade. Os comandos de atribuição, em uma linguagem de alto nível, podem ser decompostos em comandos mais elementares, como se vê, a seguir:

Processo A	Processo B
LOAD x,Ra ADD 1,Ra STORE Ra,x	LOAD x,Rb SUB 1,Rb STORE Rb,x

Considere que o Processo A carregue o valor de X no registrador Ra, some 1 ao registrador Ra, e seja interrompido no momento em que vai armazenar o valor de X. Nesse instante, o Processo B inicia sua execução, carrega o valor de X em Rb e subtrai 1 desse registrador. Dessa vez, o Processo B é interrompido e o Processo A volta a ser processado, atribuindo o valor 3 à variável X e concluindo sua execução. Finalmente, o Processo B retorna à execução atribui o valor 1 a X, e sobrepõe o valor anteriormente gravado pelo Processo A. O valor final da variável X é inconsistente em função da forma concorrente com que os dois processos executaram.

Conceitos

Condição de Corrida (*Race Condition*) é a situação na qual alguns processos acessam simultaneamente uma região crítica, tornando o resultado inconsistente e dependente da ordem em que os acessos são realizados.

Esses dois exemplos apresentam o problema do compartilhamento de recursos denominado condição de corrida (*Race Condition*). Analisando os dois exemplos apresentados, é possível concluir que em qualquer situação em que dois ou mais processos compartilham um mesmo recurso, devem existir mecanismos de controle para evitar esses tipos de problemas.

7.5. Exclusão mútua

Para evitar os problemas de compartilhamento de recursos apresentados anteriormente, deve-se impedir que dois ou mais processos acessem simultaneamente um mesmo recurso. Assim, quando um dos processos estiver acessando determinado recurso, os demais processos que queiram acessá-lo deverão aguardar pelo término da utilização do recurso. Essa ideia de exclusividade no acesso a um recurso compartilhado é chamada de **exclusão mútua** (*mutual exclusion*). A exclusão mútua deve afetar apenas os processos concorrentes e somente quando um deles estiver fazendo acesso ao recurso compartilhado [1].

Região Crítica (***Critical Region***) é a parte do código do programa onde é feito o acesso ao recurso compartilhado.

Conceitos

Se for possível evitar que dois processos entrem em suas regiões críticas ao mesmo tempo, ou seja, se for garantida a execução mutuamente exclusiva das regiões críticas, os problemas decorrentes do compartilhamento serão evitados.

Fala Professor

Os mecanismos que implementam a exclusão mútua utilizam protocolos de acesso à região crítica. Toda vez que um processo desejar executar instruções de sua região crítica, obrigatoriamente deverá executar antes um protocolo de entrada nessa região. Da mesma forma, ao sair da região crítica, um protocolo de saída deverá ser executado. Os protocolos de entrada e saída garantem a exclusão mútua da região crítica de um programa.

```
INÍCIO
        .
        .
        Entra_Região_Crítica ;     (* Protocolo de Entrada *)
        Região_Crítica ;
        Sai_Região_Crítica ;     (* Protocolo de Saída *)
        .
        .
FIM.
```

Utilizando o programa Conta_Corrente [1], apresentado no item anterior, a aplicação dos protocolos para dois processos (A e B) pode ser implementada no compartilhamento do arquivo de contas-correntes. Sempre que o Processo A for atualizar o saldo de um cliente, antes de ler o registro o acesso exclusivo ao arquivo deve ser garantido por meio do protocolo de entrada da sua região crítica. O protocolo indica se já existe ou não algum processo acessando o recurso. Caso o recurso esteja livre, o Processo A pode entrar em sua região crítica para realizar a atualização. Durante esse período, caso o Processo B tente acessar o arquivo, o protocolo de entrada faz com que esse processo permaneça aguardando até que o Processo A termine o acesso ao recurso. Quando o Processo A terminar a execução de sua região crítica, deve sinalizar aos demais processos concorrentes que o acesso ao recurso foi concluído. Isso é realizado por meio do protocolo de saída, que informa aos outros processos que o recurso já está livre e pode ser utilizado de maneira exclusiva por um outro processo.

Para garantir a implementação da exclusão mútua, os processos envolvidos devem fazer acesso aos recursos de forma sincronizada. Diversas soluções foram desenvolvidas com esse propósito; porém, para que o problema de acesso ao recurso compartilhado seja resolvido satisfatoriamente, algumas regras devem ser satisfeitas. São elas:

- Regra 1: Exclusão Mútua – dois ou mais processos não podem estar simultaneamente em uma região crítica;
- Regra 2: Progressão – nenhum processo fora da região crítica pode bloquear a execução de outro processo;
- Regra 3: Espera Limitada – nenhum processo deve esperar infinitamente (*starvation*) para entrar em uma região crítica;
- Regra 4: Independência da quantidade de processadores – a solução não deve fazer considerações sobre o número de processadores nem de suas velocidades relativas.

Ao satisfazer a regra 2, a solução evita uma situação indesejada na implementação da exclusão mútua em que um processo fora de sua região crítica impede que outros processos entrem nas suas próprias regiões críticas. No caso de esta situação ocorrer, um recurso estaria livre, porém alocado a um processo. Com isso, vários processos estariam sendo impedidos de utilizar o recurso, reduzindo o grau de compartilhamento.

Ao satisfazer a regra 3, a solução evita outra situação indesejada que é

conhecida como *starvation* ou **espera indefinida**. *Starvation* é a situação em que um processo nunca consegue executar sua região crítica e, consequentemente, acessar o recurso compartilhado. No momento em que um recurso alocado é liberado, o sistema operacional possui um critério para selecionar, entre os processos que aguarda pelo uso do recurso, qual será o escolhido. Em função do critério de escolha, o problema do *starvation* pode ocorrer. Os critérios de escolha aleatória ou com base em prioridades são exemplos de situações que podem gerar o problema. No primeiro caso, não é garantido que um processo, em algum momento, fará o uso do recurso, pois a seleção é randômica. No segundo caso, a baixa prioridade de um processo em relação aos demais pode impedir o processo de acessar o recurso compartilhado. Uma solução bastante simples para o problema é a criação de filas de pedidos de alocação para cada recurso, utilizando o esquema FIFO. Sempre que um processo solicita um recurso, o pedido é colocado no final da fila associada ao recurso. Quando o recurso é liberado, o sistema seleciona o primeiro processo da fila.

Ao satisfazer a regra 4, a solução garante seu funcionamento, independente da quantidade de processadores e de suas velocidades relativas. Uma solução que funcione em ambientes com um único processador, mas não funcionem com dois ou mais processadores, não é considerada uma boa solução.

Entre as diversas soluções propostas para garantir a exclusão mútua de processos concorrentes, estão as de hardware e software, como:

- Desabilitação de Interrupções;
- Variáveis do tipo *lock;*
- Alternância;
- Semáforos;
- Monitores.

Nem todas as soluções acima satisfazem as quatro regras necessárias para serem consideradas boas soluções. Por exemplo, “Desabilitação de Interrupções” funciona apenas em ambientes monoprocessados, além de dar muito poder ao usuário para ele desabilitá-las, pois somente ao sistema operacional deve ser permitido desabilitar as interrupções; “Variáveis do tipo *Lock*” apresenta condições de corrida, além de espera ocupada (ao invés de ficar no estado bloqueado, o processo que está esperando sua vez de entrar na região crítica fica ocupando o processador, realizando testes sucessivos); a solução “Alternância” fere a regra 2, pois um processo fora

da região crítica pode bloquear um outro processo. Já as soluções que utilizam "Semáforos" e as que utilizam "Monitores" satisfazem todas as regras. Neste material será apresentado como solução para o problema do compartilhamento de recursos apenas os "Semáforos".

7.6. Sincronização condicional

Conceitos

Sincronização condicional é uma situação em que o acesso ao recurso compartilhado exige a sincronização de processos vinculada a uma condição de acesso. Um recurso pode não se encontrar pronto para uso, devido a uma condição específica. Nesse caso, o processo que deseja acessá-lo deverá permanecer bloqueado até que o recurso fique disponível.

Um exemplo clássico desse tipo de sincronização é a comunicação entre dois processos por meio de operações de gravação e leitura em um buffer, como visto anteriormente, em que processos geram informações (**processos produtores**) utilizadas por outros processos (**processos consumidores**). Nessa comunicação, enquanto um processo grava dados em um buffer, o outro lê os dados, concorrentemente. Os processos envolvidos devem estar sincronizados a uma variável de condição, de forma que um processo não tente gravar dados em um buffer cheio ou realizar uma leitura em um buffer vazio.

Fala Professor

No exemplo **produtor-consumidor**, *se o buffer ficar cheio, o produtor não poderá produzir até que o consumidor consuma algum dado, liberando alguma posição; se o buffer ficar vazio, o consumidor não poderá consumir até que um novo dado seja colocado no buffer pelo produtor.*

Assim, quando o buffer fica cheio, o produtor coloca-se para dormir (entra em estado de espera) e somente será acordado (retirado do estado de espera) pelo consumidor, quando este liberar uma posição do buffer. Se o buffer ficar vazio, o consumidor coloca-se para dormir e somente será acordado pelo produtor quando uma nova posição do buffer for preenchida.

Atividades

1. Defina o que é uma aplicação concorrente e dê um exemplo de sua utilização.
2. Considere uma aplicação que utilize uma matriz na memória principal para a comunicação entre vários processos concorrentes. Que tipo de problema pode ocorrer quando dois ou mais processos acessam uma mesma posição da matriz?
3. Explique o que é exclusão mútua e como é implementada.
4. Explique é *starvation* e como podemos solucionar esse problema.
5. Defina o que é sincronização condicional e dê um exemplo de sua utilização.

7.7. Semáforos

O conceito de semáforos foi proposto por E.W.Dijkstra, em 1965, como um mecanismo de sincronização que permite implementar, de forma simples, a exclusão mútua e a sincronização condicional entre processos. De fato, o uso de semáforos tornou-se um dos principais mecanismos utilizados em projetos de sistemas operacionais e em aplicações concorrentes. Atualmente, a maioria das linguagens de programação disponibiliza rotinas para uso de semáforos.

Um semáforo é uma variável inteira, não-negativa, que só pode ser manipulada por duas instruções: DOWN e UP, também chamadas, originalmente, por Dijkstra, de intruções P (*proberen*, teste, em holandês) e V (*verhogen*, incremento, em holandês). As instruções DOWN e UP devem ser indivisíveis (ou atômicas), ou seja, a execução dessas instruções não pode ser interrompida. A instrução UP incrementa uma unidade ao valor do semáforo, enquanto a DOWN decrementa uma unidade. Visto que, por definição, valores negativos não podem ser atribuídos a um semáforo, se a instrução DOWN executa em um semáforo com valor 0, o processo entrará no estado de espera. Em geral, o processador deverá dar suporte à implementação dessas instruções, para que todas essas condições sejam garantidas.

Os semáforos podem ser classificados como **binários** ou **contadores**. Os semáforos binários, também chamados de **mutexes** (*mutual exclusion semaphores*), só podem assumir os valores 0 e 1, enquanto os semáforos contadores podem assumir qualquer valor inteiro positivo, além de 0.

Fala Professor

A implementação da **exclusão mútua** *é realizada por meio de* **semáforos binários** *(mutex), ao passo que a* **sincronização condicional** *é realizada por meio de* **semáforos contadores**.

7.7.1 - Exclusão mútua utilizando semáforos

Como visto anteriormente, a exclusão mútua pode ser implementada por meio de um semáforo binário, associado ao recurso compartilhado. A principal vantagem dessa solução, em relação aos algoritmos citados anteriormente, é a não ocorrência da espera ocupada. Em outras palavras, quando um processo não pode entrar na região crítica, ele é colocado para "dormir" (estado de espera), evitando ocupar o processador enquanto aguarda.

As instruções DOWN e UP funcionam como protocolos de entrada e saída, respectivamente, para que um processo possa entrar e sair de sua região crítica. O semáforo fica associado a um recurso compartilhado, indicando quando o recurso está sendo acessado por um dos processos concorrentes. O valor do semáforo igual a 1 indica que nenhum processo está utilizando o recurso, enquanto o valor 0 indica que o recurso está em uso [1].

Sempre que deseja entrar na sua região crítica, um processo executa uma instrução DOWN. Se o semáforo for igual a 1, esse valor é decrementado, e o processo que solicitou a operação pode executar as instruções da sua região crítica. De outra forma, se uma instrução DOWN for executada em um semáforo com valor igual a 0, o processo fica impedido do acesso, permanecendo numa fila em estado de espera e, consequentemente, não gerando *overhead* no processador.

O processo que está acessando o recurso, ao sair de sua região crítica, executa uma instrução UP, e, se nenhum processo estiver aguardando a utilização do recurso, o valor do semáforo será incrementado, liberando o acesso ao recurso. Porém, se houver um ou mais processos esperando pelo uso do recurso (operações DOWN pendentes), o sistema selecionará um processo na fila de espera associada ao recurso e alterará o seu estado para pronto, permitindo a entrada desse outro processo na região crítica.

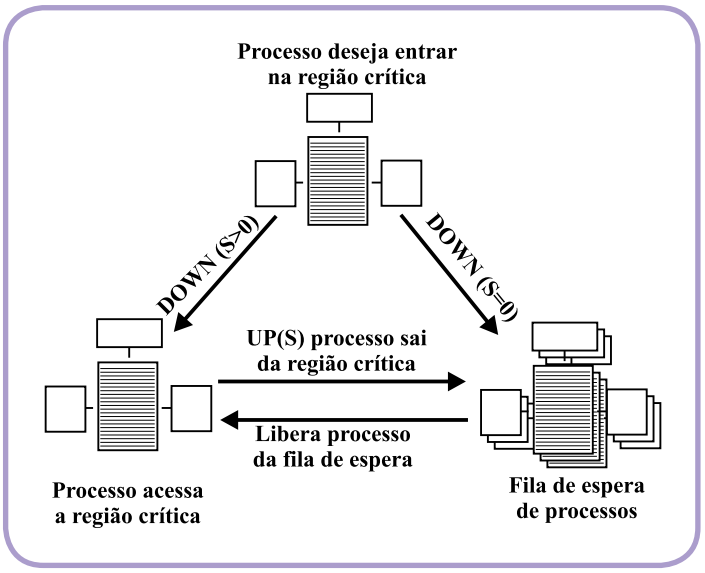

Figura 7-54:Utilização do semáforo binário na exclusão mútua
Fonte: [1] – Machado e Maia, 2004. Adaptação.

As instruções DOWN e UP, aplicadas a um semáforo S, podem ser representadas pelas definições a seguir, em uma sintaxe Portugol não convencional:

```
TIPO Semáforo = REGISTRO

      Valor : INTEIRO;

      Fila_Espera : FILA; (* Fila de processos pen-
dentes *)
      FIM.

PROCEDIMENTO DOWN (VAR S: Semáforo)
INÍCIO
      SE (S.Valor = 0) ENTÃO

      Coloca_Processo_na_Fila_de_Espera
      ELSE

      S.Valor := S.Valor - 1;
FIM;
PROCEDIMENTO UP(VAR S: Semáforo)
INÍCIO
      SE (Tem_Processo_na_Fila_de_Espera) ENTÃO

      Retira_Primeiro_da_Fila_de_Espera
      ELSE

      S.Valor := S.Valor + 1;
FIM;
```

O programa Semáforo_1 apresenta uma solução para o problema de exclusão mútua entre dois processos, utilizando semáforos. O semáforo é inicializado com o valor 1, indicando que nenhum processo está executando sua região crítica.

```
PROGRAMA Semaforo_1;
      VAR S : SEMÁFORO := 1; (* inicialização do se-
máforo *)

PROCEDIMENTO Processo_A;
INÍCIO
      REPITA

      DOWN (S);

      Região_Crítica_A;

      UP(S);
      ATÉ Falso;
FIM;

PROCEDIMENTO Processo_B;
INÍCIO
      REPITA

      DOWN (S);

      Região_Crítica_B;

      UP(S);
      ATÉ Falso;
FIM;

INÍCIO
      PARBEGIN (* inicia os processos A e B concor-
rentemente *)

      Processo_A;

      Processo_B;
      PAREND;
FIM.
```

Nesse exemplo, o primeiro processo que executar a primitiva DOWN obterá acesso a sua região crítica. Quando o segundo processo executar a primitiva DOWN, ficará bloqueado até que o primeiro libere o acesso através da execução da primitiva UP. Após isso, o segundo processo poderá entrar em sua região crítica. Quando o segundo processo executar um UP, o valor do semáforo será incrementado – pois não haverá mais

nenhum processo na fila – e voltará ao valor inicial.

7.7.2. Sincronização condicional utilizando semáforos

Além de permitir a implementação da exclusão mútua, os semáforos podem ser utilizados nos casos em que a sincronização condicional é exigida. Um exemplo desse tipo de sincronização ocorre quando um processo solicita uma operação de E/S. O pedido faz com que o processo execute uma instrução DOWN no semáforo associado ao evento e fique no estado de espera, até que a operação seja completada. Quando a operação termina, a rotina de tratamento da interrupção executa um UP no semáforo, liberando o processo do estado de espera.

O problema do produtor/consumidor, apresentado anteriormente, é um exemplo de como a exclusão mútua e a sincronização condicional podem ser implementadas por meio de semáforos. O programa Produtor_Consumidor [1] apresenta uma solução para esse problema, utilizando um buffer de apenas cinco posições. O programa utiliza três semáforos, sendo um do tipo binário, utilizado para implementar a exclusão mútua, e dois semáforos contadores, para a sincronização condicional. O semáforo binário Mutex permite a execução das regiões críticas Grava_Buffer e Le_Buffer de forma mutuamente exclusivas. Os semáforos contadores: "Vazias" e "Cheias" representam, respectivamente, se há posições livres no buffer para serem gravadas e posições ocupadas a serem lidas. O semáforo Vazias - igual a 0, significa que o buffer está cheio e o processo produtor deverá aguardar até que o consumidor leia algum dado. Da mesma forma, o semáforo Cheias - igual a 0, significa que o buffer está vazio e o consumidor deverá aguardar até que o produtor grave algum dado. Em ambas as situações, o semáforo sinaliza a impossibilidade de acesso ao recurso.

```
PROGRAMA Produtor_Consumidor;
      CONST TamBuf = 5;
      TIPO Tipo_Dado = (* Tipo qualquer *);
      VAR Vazias : Semáforo := TamBuf;
          Cheias : Semáforo := 0;
          Mutex  : Semáforo := 1;
          Buffer : VETOR [1..TamBuf] DE Tipo_Dado;
          Dado_1 : Tipo_Dado;
          Dado_2 : Tipo_Dado;

PROCEDIMENTO Produtor;
INÍCIO
      REPITA

      Produz_Dado (Dado_1);
```

```
      DOWN (Vazias);

      DOWN (Mutex);

      Grava_Buffer (Dado_1, Buffer);

      UP (Mutex);

      UP (Cheias);
      ATÉ Falso;
FIM;

PROCEDIMENTO Consumidor;
INÍCIO
      REPITA

      DOWN (Cheias);

      DOWN (Mutex);

      Le_Buffer (Dado_2, Buffer);

      UP (Mutex);

      UP (Vazias);

      Consome_Dado (Dado_2);
      ATÉ Falso;
FIM;
INÍCIO
      PARBEGIN   (* inicia processos concorrentemen-
te *)

      Produtor;

      Consumidor;
      PAREND;

FIM.
```

O exemplo anterior considera que o processo Consumidor é o primeiro a ser executado. Como o buffer está vazio (Cheias. Valor=0), a operação DOWN (Cheias) faz com que o processo Consumidor permaneça no estado de espera (bloqueado). Em seguida, o processo Produtor grava um dado no buffer e, após executar o UP (Cheias), libera o processo Consumidor, que estará apto para ler o dado do buffer.

Semáforos contadores são bastante úteis, quando aplicados em problemas de sincronização condicional, em que existem processos concorrentes alocando recursos do mesmo tipo (pool de recursos). O semáforo

é inicializado com o número total de recursos do pool, e, sempre que um processo deseja alocar um recurso, executa um DOWN, subtraindo 1 do número de recursos disponíveis. Da mesma forma, sempre que o processo libera um recurso para o pool, executa um UP. Se o semáforo contador ficar com o valor igual a 0, não existirão mais recursos a serem utilizados e o processo que solicita um recurso, permanece em estado de espera, até que outro processo libere algum recurso para o pool.

7.8. Deadlock

Deadlock é a situação em que um processo aguarda por um recurso que nunca estará disponível ou um evento que não ocorrerá. Essa situação é consequência, na maioria das vezes, do compartilhamento de recursos, como dispositivos, arquivos e registros, entre processos concorrentes em que a exclusão mútua é exigida. O problema do deadlock torna-se cada vez mais frequente e crítico na medida em que os sistemas operacionais evoluem no sentido de implementar o paralelismo de forma intensiva e permitir a alocação dinâmica de um número ainda maior de recursos [1].

Por exemplo, suponha que existam dois processos, PA e PB, e ambos utilizam os recursos R1 e R2. Inicialmente, PA obtém acesso exclusivo de R1, da mesma forma que PB obtém de R2. Durante o processamento, PA necessita de R2 para poder prosseguir. Como R2 está alocado a PB, PA ficará aguardando que o recurso seja liberado. PB, por sua vez, necessita utilizar R1 e, da mesma forma, ficará aguardando até que PA libere o recurso. Como cada processo está esperando que o outro libere o recurso alocado, é estabelecida uma condição conhecida como espera circular, caracterizando uma situação de deadlock.

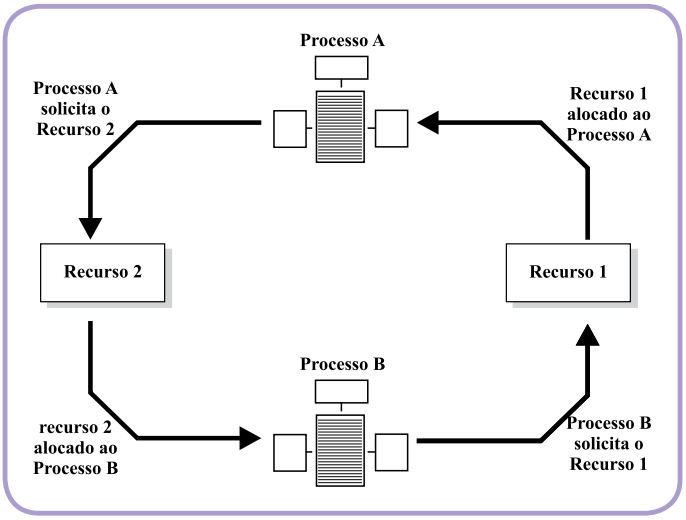

Figura 7-55: Exemplo de espera circular
Fonte: [1] – Machado e Maia, 2004. Adaptação.

Para que ocorra a situação de deadlock, quatro condições são necessárias simultaneamente:

1. **Exclusão mútua**: cada recurso só pode estar alocado a um único processo, em um determinado instante;
2. **Espera por recurso**: um processo, além dos recursos já alocados, pode estar esperando por outros recursos;
3. **Não-preempção**: um recurso não pode ser liberado de um processo só porque outros processos desejam o mesmo recurso;
4. **Espera circular**: um processo pode ter de esperar por um recurso alocado ao outro processo e vice-versa.

O problema do deadlock existe em qualquer sistema multiprogramável; no entanto, as soluções implementadas devem considerar o tipo do sistema e o impacto em seu desempenho. Por exemplo, um deadlock em sistema de tempo real, que controla uma usina nuclear, deve ser tratado com mecanismos voltados para esse tipo de aplicação, diferentes dos adotados por um sistema de tempo compartilhado comum.

7.8.1. Prevenção de deadlock

Para prevenir a ocorrência de deadlocks, é preciso garantir que uma das quatro condições apresentadas, necessárias para sua existência, nunca se satisfaça.

A ausência da primeira condição (exclusão mútua), certamente, acaba com o problema do deadlock, pois nenhum processo terá que esperar para ter acesso a um recurso, mesmo que já esteja sendo utilizado por outro processo. No entanto, os problemas decorrentes do compartilhamento de recursos entre processos concorrentes sem a exclusão mútua já foram apresentados, inclusive, com a exemplificação das sérias inconsistências geradas. Portanto, não é possível eliminar a primeira condição devido à necessidade de se obter a exclusão mútua.

Para evitar a segunda condição (espera por recurso), processos que já possuam recursos garantidos não devem requisitar novos recursos. Uma maneira de implementar esse mecanismo de prevenção é que, antes do início da execução, um processo deve pré-alocar todos os recursos necessários. Nesse caso, todos os recursos necessários ao processo devem estar disponíveis para o início da execução, caso contrário, nenhum recurso será alocado e o processo permanecerá aguardando. Esse mecanismo pode produzir um grande desperdício na utilização dos recursos do sistema, pois um recurso pode permanecer alocado por um grande período de tempo,

sendo utilizado apenas por um breve momento. Outra dificuldade, decorrente desse mecanismo, é a determinação do número de recursos que o processo deve alocar antes da sua execução. Mais grave, no entanto, é a possibilidade de um processo sofrer *starvation*, caso os recursos necessários à sua execução nunca estejam disponíveis ao mesmo tempo.

A terceira condição (não-preempção) pode ser evitada, quando é permitido que um recurso seja retirado de um processo, no caso de outro processo necessitar do mesmo recurso. A liberação de recursos já garantidos por um processo pode ocasionar sérios problemas, podendo até impedir a continuidade de sua execução. Outro problema desse mecanismo é a possibilidade de um processo sofrer *starvation*, pois, após garantir os recursos necessários à sua execução, pode ter que liberá-los, sem concluir seu processamento.

A última maneira de se evitar um deadlock é excluir a possibilidade da quarta condição (espera circular). Uma forma de se implementar esse mecanismo é forçar o processo a ter apenas um recurso por vez. Caso o ele necessite de outro recurso, o recurso já alocado deve ser liberado. Essa condição restringiria muito o grau de compartilhamento e o processamento de programas.

A prevenção de deadlocks, evitando-se a ocorrência de qualquer uma das quatro condições, é bastante limitada e, por isso, não é utilizada na prática. É possível evitar o deadlock, mesmo se todas as condições necessárias à sua ocorrência estiverem presentes. A solução mais conhecida para essa situação é o Algoritmo do Banqueiro (*Banker's Algorithm*) proposto por Dijkstra (1965). Basicamente, esse algoritmo exige que os processos informem o número máximo de cada tipo de recurso necessário à sua execução. Com essas informações, é possível definir o estado de alocação de um recurso, que é a relação entre um número de recursos alocados e disponíveis e o número máximo de processos que necessitam desses recursos. Com base nessa relação, o sistema pode fazer a alocação dos recursos de forma segura e evitar ocorrência de deadlocks.

Apesar de evitar o problema do deadlock, essa solução possui também várias limitações. A maior delas é a necessidade de um número fixo de processos ativos e de recursos disponíveis no sistema. Essa limitação impede que a solução seja implementada na prática, pois é muito difícil prever o número de usuários no sistema e um número de recursos disponíveis.

7.8.2. Detecção do deadlock

Em sistemas que não possuam mecanismos que previnam a ocorrên-

cia de deadlocks, é necessário um esquema de detecção e correção do problema. A detecção do deadlock é o mecanismo que determina, realmente, a existência da situação de deadlock, permitindo identificar os recursos e processos envolvidos no problema.

Para detectar deadlocks, os sistemas operacionais devem manter estruturas de dados capazes de identificar cada recurso de sistema, o processo que o está alocando e os processos que estão à espera da liberação de recurso. Toda vez que um recurso é alocado ou liberado por um processo, a estrutura deve ser atualizada. Geralmente, os algoritmos que implementam esse mecanismo verificam a existência da espera circular, percorrendo toda a estrutura, sempre que um processo solicita um recurso e ele não pode ser imediatamente garantido.

Dependendo do tipo de sistema, o ciclo de busca por um deadlock pode variar. Em sistemas de tempo compartilhado, o tempo de busca pode ser maior, sem comprometer o desempenho e a confiabilidade do sistema. Sistemas de tempo real, por sua vez, devem constantemente se certificar da ocorrência de deadlocks; porém, a maior segurança gera maior overhead.

7.8.3. Correção do deadlock

Após a detecção de deadlock, o sistema operacional deverá, de alguma forma, corrigir o problema. Uma solução bastante utilizada pela maioria dos sistemas é eliminar um ou mais processos envolvidos no deadlock e desalocar os recursos já garantidos por eles, quebrando, assim, a espera circular.

A eliminação dos processos envolvidos no deadlock e, consequentemente a liberação dos seus recursos, podem não ser simples, dependendo do tipo de recurso envolvido. Se um processo estiver atualizando um arquivo ou imprimindo uma listagem, o sistema deve garantir que esses recursos sejam liberados sem problemas. Os problemas eliminados não têm como ser recuperados; porém, outros processos, que antes estavam em deadlock, poderão conseguir a execução.

A escolha do processo a ser eliminado é feita, normalmente, de forma aleatória ou, então, com base em algum tipo de prioridade. Esse esquema, no entanto, pode consumir considerável tempo do processador, gerando elevado overhead ao sistema.

Uma solução menos drástica envolve a liberação de apenas alguns recur-

sos alocados aos processos para outros processos, até que o ciclo de espera termine. Para que essa solução seja, realmente, eficiente, é necessário que o sistema possa suspender um processo, liberar seus recursos e, após a solução do problema, retornar à execução do processo, sem perder o processamento já realizado. Esse mecanismo é conhecido como *rollback* e (além do overhead gerado) e é muito difícil de ser implementado, por ser bastante dependente da aplicação que está sendo processada.

Indicações

[1] MACHADO, F.B. e MAIA, L.P. ***Arquitetura de Sistemas Operacionais***. 4.ed. LTC, 2007.

[2] SILBERSCHATZ, A., GALVIN, P.B., GAGNE, G. ***Fundamentos de Sistemas Operacionais***. 6.ed. LTC, 2004.

[3] TANENBAUM, A.S. ***Sistemas Operacionais Modernos***. 2.ed. Pearson Brasil, 2007.

[4] OLIVEIRA, R.S., CARISSIMI, A.S., TOSCANI, S.S. ***Sistemas Operacionais***. 3.ed. Sagra-Luzzato. 2004.

Atividades

Atividades

1. Pesquise no livro-texto, qual o problema com a solução que de sabilita as interrupções para implementar a exclusão mútua.
2. Pesquise no livro-texto, o que é **espera ocupada** e qual o seu problema?
3. Explique o que são semáforos e dê dois exemplos de sua utilização: um para a solução da exclusão mútua e outro para a sincronização condicional.
4. O que é deadlock, quais as condições para obtê-lo e quais as soluções possíveis?
5. Em uma aplicação concorrente que controla saldo bancário em contas correntes, dois processos compartilham uma região de memória onde estão armazenados os saldos dos clientes A e B. Os processos executam, concorrentemente, os seguintes passos:

Processo 1 (Cliente A)	Processo 2 (Cliente B)
/* saque em A */	/*saque em A */

Atividades

1a. x := saldo_do_cliente_A;	2a. y := saldo_do_cliente_A;
1b. x := x - 200;	2b. y := y - 100;
1c. saldo_do_cliente_A := x;	2c. saldo_do_cliente_A := y;
/* deposito em B */	/* deposito em B */
1d. x := saldo_do_cliente_B;	2d. y := saldo_do_cliente_B;
1e. x := x + 100;	2e. y := y + 200;
1f. saldo_do_cliente_B := x;	2f. saldo_do_cliente_B := y;

6. Supondo que os valores dos saldos de A e B sejam, respectivamente, 500 e 900, antes de os processos executarem, pede-se:

a) Quais os valores corretos esperados para os saldos dos clientes A e B, após o término da execução dos processos?

b) Quais os valores finais dos saldos dos clientes, se a sequência temporal de execução das operações for: 1a, 2a, 1b, 2b, 1c, 2c, 1d, 2d, 1e, 2e, 1f, 2f?

c) Utilizando semáforos, proponha uma solução que garanta a integridade dos saldos e permita o maior compartilhamento possível dos recursos entre os processos, não esquecendo a especificação da inicialização dos semáforos.

7. O problema dos leitores/escritores, apresentado a seguir, consiste em sincronizar processos que consultam/atualizam dados em uma base comum. Pode haver mais de um leitor lendo ao mesmo tempo; no entanto, enquanto um escritor está atualizando a base, nenhum outro processo pode ter acesso a ela (nem mesmo leitores).

```
VAR  Acesso: Semaforo := 1;
     Exclusao: Semaforo := 1;
     Nleitores: integer := 0;
PROCEDURE Escritor;
BEGIN
    ProduzDado;
    DOWN (Acesso);
    Escreve;
    UP (Acesso);
END;
```

Atividades

```
PROCEDURE Leitor;
BEGIN
    DOWN (Exclusao);
    Nleitores := Nleitores + 1;
    IF ( Nleitores = 1 ) THEN DOWN (Acesso);
    UP (Exclusao);
    Leitura;
    DOWN (Exclusao);
    Nleitores := Nleitores - 1;
    IF ( Nleitores = 0 ) THEN UP (Acesso);
    UP (Exclusao);
    ProcessaDado;
END;
```

a. Suponha que exista apenas um leitor fazendo acesso à base. Enquanto esse processo realiza a leitura, quais os valores das três variáveis?

b. Chega um escritor, enquanto o leitor ainda está lendo. Quais os valores das três variáveis, após o bloqueio do escritor? Sobre qual(is) semáforo(s) se dá o bloqueio?

c. Chega mais um leitor, enquanto o primeiro ainda não acabou de ler, e o escritor está bloqueado. Descreva os valores das três variáveis, quando o segundo leitor inicia a leitura.

d. Os dois leitores terminam simultaneamente a leitura. É possível haver problemas quanto à integridade do valor da variável **nleitores**? Justifique.

e. Descreva o que acontece com o escritor, quando os dois leitores terminam suas leituras. Descreva os valores das três variáveis, quando o escritor inicia a escrita.

f. Enquanto o escritor está atualizando a base, chega mais um escritor e mais um leitor. Sobre qual(is) semáforo(s) eles ficam bloqueados? Descreva os valores das três variáveis, após o bloqueio dos recém-chegados.

g. Quando o escritor tiver terminado a atualização, é possível prever qual dos processos bloqueados (leitor ou escritor) terá acesso primeiro à base?

h. Descreva uma situação em que os escritores sofram starvation (adiamento indefinido).